Interface Graphique en Java 1.6

Événements

Sébastien Paumier

L'EventDispatchThread

- alias "la thread Swing": thread qui pompe et traite les d'événements d'une queue d'événements
- les événements traités viennent:
  - du système (fermeture de l'application, iconification d'une fenêtre, rafraîchissement d'une zone qui redevient visible, etc)
  - de l'utilisateur (souris, clavier)
  - de l'application (tâches périodiques, etc)

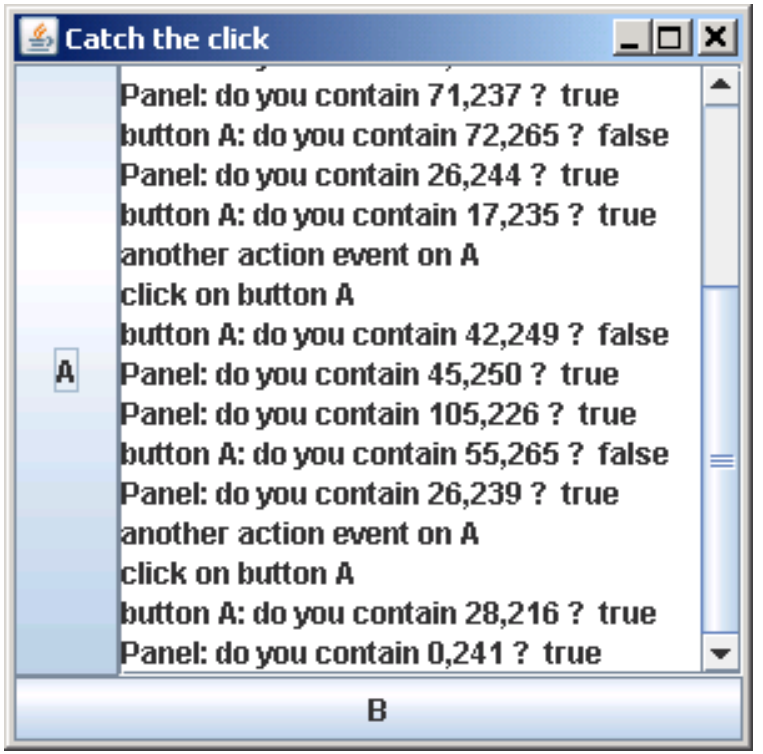
L'EventDispatchThread

- les événements sont dispatchés aux objets concernés
- exemple: qui est concerné par un clic ?
  - Swing inspecte l'arborescence de composants pour trouver qui contient le clic, de façon récursive, en tenant compte des positions de chacun
  - sur le composant concerné, la VM exécute `processMouseEvent`, afin d'exécuter tous les bouts de code associés à cet événement

Les listeners

- pour réagir à des événements on utilise des listeners, éventuellement plusieurs pour un même objet

```
a.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        myModel.addElement("click on button A");
        list.ensureIndexIsVisible(myModel.getSize()-1);
    }
});
a.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        myModel.addElement("another action event on A");
        list.ensureIndexIsVisible(myModel.getSize()-1);
    }
});
```


Catch the click
Panel: do you contain 71,237 ? true
button A: do you contain 72,265 ? false
Panel: do you contain 26,244 ? true
button A: do you contain 17,235 ? true
another action event on A
click on button A
button A: do you contain 42,249 ? false
A
Panel: do you contain 45,250 ? true
Panel: do you contain 105,226 ? true
button A: do you contain 55,265 ? false
Panel: do you contain 26,239 ? true
another action event on A
click on button A
button A: do you contain 28,216 ? true
Panel: do you contain 0,241 ? true
B

Principe des listeners

1) les objets qui ont des "choses à dire" proposent l'ajout/retrait de listeners:
   - `JButton`: `add`/`removeActionListener`
   - `JInternalFrame`: `add`/`removeInternalFrameListener`
   - etc
- un listener est une interface proposant des méthodes spécialisées:
   - `ActionListener`: `actionPerformed`
   - `InternalFrameListener`: `internalFrameClosing`
   - etc

Principe des listeners

2) les objets qui veulent être tenus au courant ajoutent un listener

3) quand il y a "quelque chose à dire", l'objet source invoque la méthode appropriée de tous les listeners qu'il connaît: c'est le travail de **`fire...`**

```
ArrayList<ActionListener> listeners=new ArrayList<ActionListener>();

/* A fire method MUST be protected, because the source object
 * must be the only one able to tell about changes */
protected void fireActionPerformed(ActionEvent event) {
   for (ActionListener l:listeners) {
   l.actionPerformed(event);
}
}
```

Des listeners personnalisés

- exemple: un listener pour être prévenu quand le texte d'un bouton devient rouge
- étape 1: définir le listener et le type d'événement qu'il doit gérer

```
public interface RedForegroundListener {

    public void redForegroundSet(RedForegroundEvent event);

}
```

```
public class RedForegroundEvent extends EventObject {

    private final Color previousColor;

    public RedForegroundEvent(Object source,Color c) {
        super(source);
        this.previousColor=c;
    }

    public Color getPreviousColor() {
        return previousColor;
    }
}
```

Des listeners personnalisés

- étape 2: fabriquer une version de **`JButton`** qui propose ce service
- gestion des listeners:

```
public class SpecialButton extends JButton {

...

    private final ArrayList<RedForegroundListener> listeners = new ArrayList<RedForegroundListener>();
    public void addRedForegroundListener(RedForegroundListener l) {
        listeners.add(l);
    }
    public void removeRedForegroundListener(RedForegroundListener l) {
        if (firingRedForegroundSet)
            throw new IllegalStateException("Cannot remove listeners while fire... is using them");
        listeners.remove(l);
    }
    protected boolean firingRedForegroundSet = false;
}
```

pas de suppression de listener pendant un **`fire...`**

Des listeners personnalisés

- comme `fire...` n'est exécutée que par une seule thread, un verrou booléen suffit:

```
protected boolean firingRedForegroundSet = false;
protected void fireRedForegroundSet(Color previous) {
    try {
        firingRedForegroundSet = true;
        RedForegroundEvent e = new RedForegroundEvent(this, previous);
        for (RedForegroundListener l : listeners) {
            l.redForegroundSet(e);
        }
    } finally {
        /* Very important to avoid being blocked because of an
         * exception raised in a listener implementation */
        firingRedForegroundSet = false;
    }
}
```

Des listeners personnalisés

- redéfinition du constructeur pour gérer l'événement qui nous intéresse:

là, on veut vraiment changer le comportement de **JButton**

```
public class SpecialButton extends JButton {

    Color previousColor;

    public SpecialButton(String text) {
        super(text);
        previousColor=getForeground();
        addPropertyChangeListener("foreground",new PropertyChangeListener() {
            @Override
            public void propertyChange(PropertyChangeEvent evt) {
                if (Color.RED.equals(getForeground())) {
                    fireRedForegroundSet(previousColor);
                }
                previousColor=getForeground();
            }
        });
    }
```

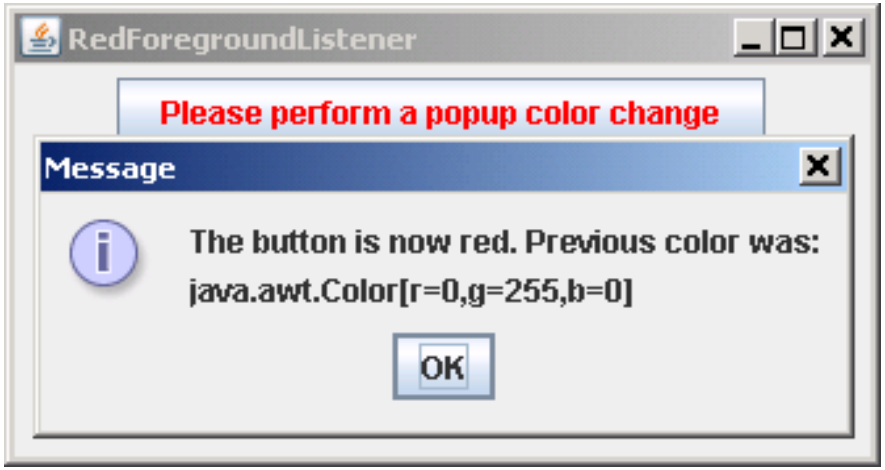
on écoute les changements de couleur de texte: si la couleur devient rouge, on va alors lever notre propre événement

Des listeners personnalisés

- étape 3: créer un bouton et mettre un listener dessus

```
final SpecialButton b=new SpecialButton("Please perform a popup color change");
b.addRedForegroundListener(new RedForegroundListener() {
    @Override
    public void redForegroundSet(RedForegroundEvent event) {
        JOptionPane.showMessageDialog((Component)event.getSource(),
            "The button is now red. Previous color was:\n"+event.getPreviousColor());
        }
});
```

- étape 4: savourer le plus beau et utile listener du monde :)



Écrire de beaux listeners

- conseil n°1: si le listener ne sert que sur un seul composant, utiliser une classe anonyme (elles ont été créées dans ce but)

```
final SpecialButton b=new SpecialButton("Please perform a popup color change");
b.addRedForegroundListener(new RedForegroundListener() {
    @Override
    public void redForegroundSet(RedForegroundEvent event) {
        JOptionPane.showMessageDialog((Component)event.getSource(),
            "The button is now red. Previous color was:\n"+event.getPreviousColor());
        }
});
```

Écrire de beaux listeners

- le listener peut accéder aux champs de la classe, ainsi qu'aux variables locales finales

```
public class Accessibility {

    int n;

    @SuppressWarnings("serial")
    public static void main(String[] args) {
        final JFrame f = new JFrame("Counter=0");
        JButton b=new JButton("Click to increase the frame title counter");
        final Accessibility accessibility=new Accessibility();
        b.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                accessibility.n++;
                f.setTitle("Counter="+accessibility.n);
            }
        });
        f.getContentPane().add(b);
        f.setSize(300,300);
        f.setDefaultCloseOperation(args.length == 0 ? JFrame.EXIT_ON_CLOSE
                                   : WindowConstants.DISPOSE_ON_CLOSE);
        f.setVisible(true);
    }

}
```

Écrire de beaux listeners

- pourquoi **`final`** ?
- parce qu'à l'exécution du code du listener, on est sorti de la fonction **`main`** où était déclarée **`f`**

```
final JFrame f = new JFrame("Counter=0");
JButton b=new JButton("Click to increase the frame title counter");
final Accessibility accessibility=new Accessibility();
b.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        accessibility.n++;
        f.setTitle("Counter="+accessibility.n);
    }
});
```

code qui sera exécuté on ne sait pas quand dans la thread Swing

Écrire de beaux listeners

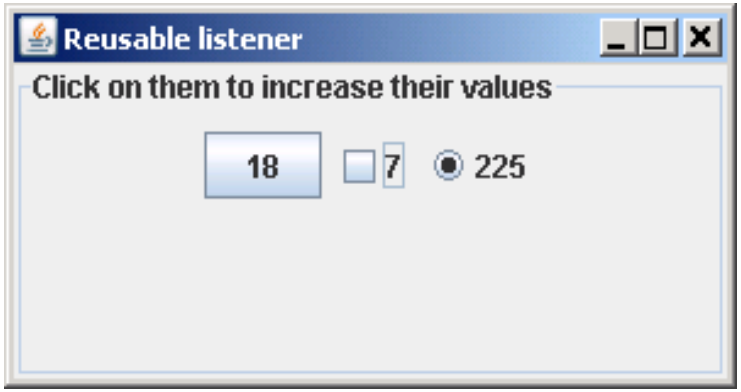
- conseil n°2: si le listener doit accéder à beaucoup d'infos, créer une classe

```
public class ShiftColorListener implements ActionListener {

    private final JComponent[] components;
    private final JFrame f;
    private final int length;
    private int nShifts;

    public ShiftColorListener(JFrame f,JComponent[] components) {
        if (components==null || components.length==0)
            throw new IllegalArgumentException(
                "Cannot use null or empty array in ShiftColorListener\n");
        this.components=components;
        length=components.length;
        this.f=f;
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        if (components==null || length<=1) return;
        Color c=components[0].getForeground();
        for (int i=0;i<length-1;i++) {
            components[i].setForeground(components[i+1].getForeground());
        }
        components[length-1].setForeground(c);
        nShifts++;
        f.setTitle(nShifts+" color shift"+(nShifts>1?"s":""));
    }
}
```

Écrire de beaux listeners

- conseil n°3: si le même listener peut servir plusieurs fois, ne pas hésiter
- si besoin, utiliser `getSource()` pour savoir d'où vient l'événement

```
ActionListener listener=new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        AbstractButton b=(AbstractButton) e.getSource();
        int n=1+Integer.parseInt(b.getText());
        b.setText(n+"");
    }
};
JButton b=new JButton("14");
JCheckBox c=new JCheckBox("3");
JRadioButton r=new JRadioButton("222");
b.addActionListener(listener);
c.addActionListener(listener);
r.addActionListener(listener);
```



Écrire de beaux listeners

- conseil n°4: pas de test sur la source, on passe ce qu'il faut au listener

```
public class BadListener implements ActionListener {

    private final JPanel p;

    public BadListener(JPanel p) {
        this.p=p;
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        AbstractButton b=(AbstractButton) e.getSource();
        if (b.getText().equals("Red"))
            p.setBackground(Color.RED);
        else if (b.getText().equals("Blue"))
            p.setBackground(Color.BLUE);
    }
}
```

beurk!!!

1) on pourrait changer le texte

2) on pourrait vouloir d'autres couleurs

Écrire de beaux listeners

- le même en propre:

```
public class GoodListener implements ActionListener {

    private final JPanel p;
    private final Color c;

    public GoodListener(JPanel p,Color c) {
        this.p=p;
        this.c=c;
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        p.setBackground(c);
    }
}
```

listener facilement paramétrable

Les événements souris

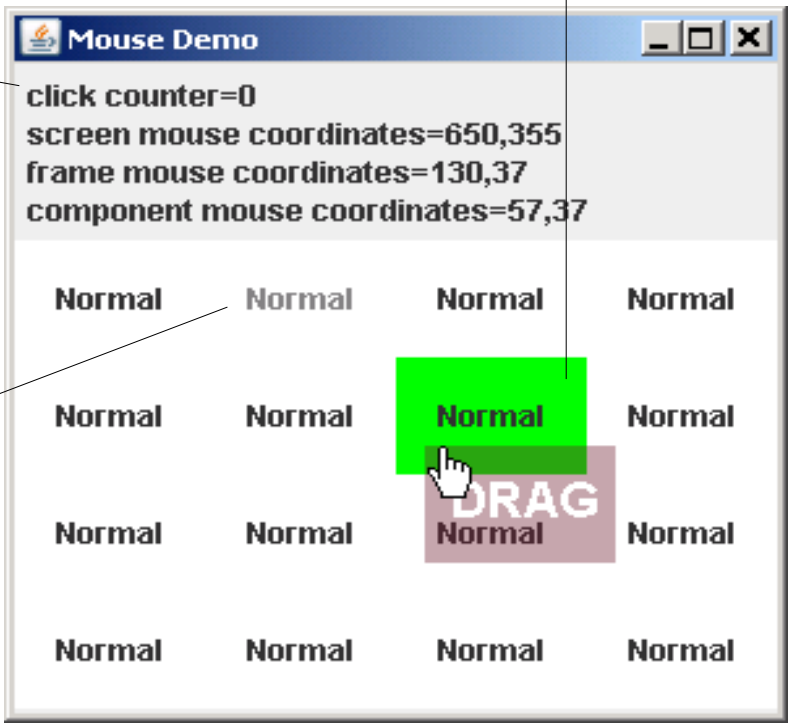
- 3 types d'événements, donc 3 listeners:
  - `MouseListener`: clic, pression, relâchement, entrée/sortie de la zone d'un composant
  - `MouseMotionListener`: déplacement de la souris
  - `MouseWheelListener`: gestion de la molette
- attention: on ne sait pas dans quel ordre `mouseReleased` et `mouseClicked` seront invoquées

Les événements souris

avec **mouseEntered**+**mouseExited**, on met en évidence le label survolé en changeant sa couleur de fond

mise à jour du compteur avec **mouseClicked**

texte du label grisé avec **mousePressed**, dégrisé avec **mouseReleased**

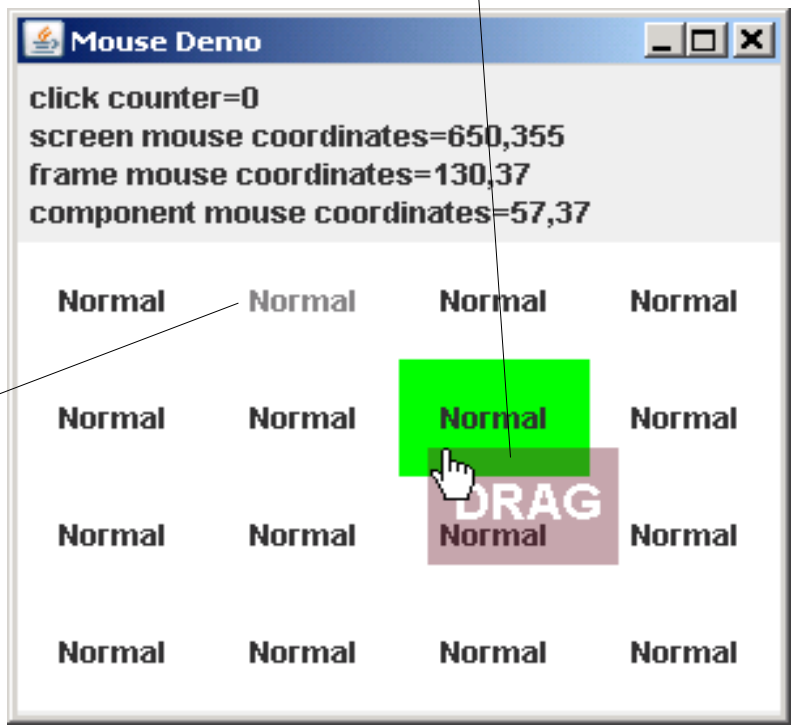


Les événements souris

avec **mouseDragged**, on change le curseur et on dessine sur le glass pane

mise à jour des coordonnées avec **mouseMoved**

grâce à **mouseReleased**, on sait quand le drag est fini pour remettre le curseur par défaut



Intermède

- pour beaucoup de **xxxListener**, il existe un **xxxAdapter** qui implémente toutes les méthodes en ne faisant rien
- pratique, car évite de s'encombrer avec les méthodes dont on ne se sert pas:

```
JLabel l=new JLabel("HELLO");
l.addMouseListener(new MouseAdapter() {
    @Override
    public void mouseClicked(MouseEvent e) {
        doSomething();
    }
});
```

PropertyChangeListener

- quand un composant modifie une de ses propriétés (couleur, opacité, etc), il prévient ceux que ça intéresse grâce aux **`PropertyChangeListener`**:

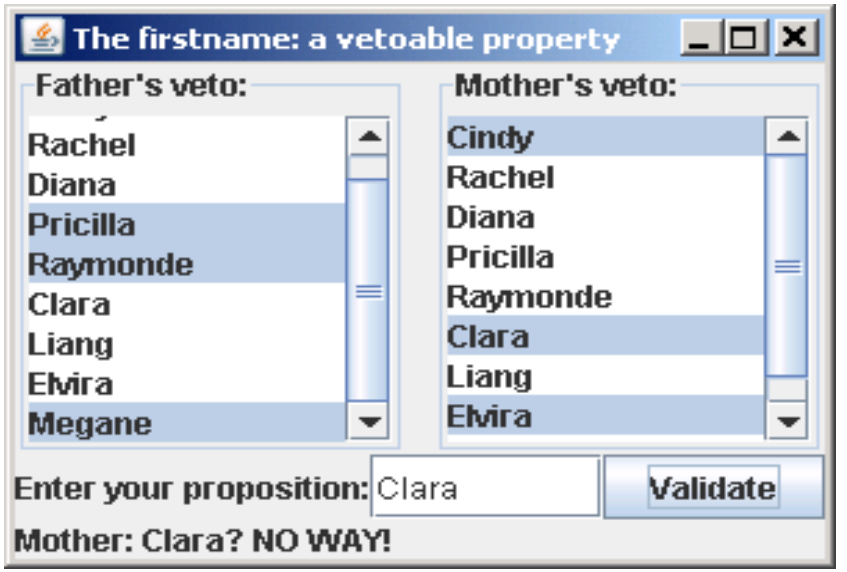
```
public class SpecialButton extends JButton {

    public SpecialButton(String text) {
        super(text);
        addPropertyChangeListener("foreground",new PropertyChangeListener() {
            @Override
            public void propertyChange(PropertyChangeEvent evt) {
                if (Color.RED.equals(getForeground())) {
                    fireRedForegroundSet();
                }
            }
        });
    }
...
}
```

on sera prévenu des changements de couleur de texte

PropertyChangeListener

- une propriété est dite *contrainte* quand elle prévient avant de changer, afin que les objets à l'écoute puissent mettre leur véto
- exemple: un label qui demande la permission avant de changer son texte

```
final JLabel firstname=new JLabel() {
    @Override
    public void setText(String text) {
        try {
            fireVetoableChange("text",getText(),text);
        } catch (PropertyVetoException e) {
            super.setText(e.getMessage());
            return;
        }
        super.setText(text);
    }
};
```



PropertyChangeListener

- pour mettre son véto, il faut s'enregistrer avec **addVetoableChangeListener** et lever une **PropertyVetoException** quand on en a envie:

```
firstname.addVetoableChangeListener(new VetoableChangeListener() {
    @Override
    public void vetoableChange(PropertyChangeEvent evt) throws PropertyVetoException {
        String s=(String) evt.getNewValue();
        for (Object o:father.getSelectedValues()) {
            if (s.equals((String)o)) {
                throw new PropertyVetoException("Father: don't even think about "+s,evt);
            }
        }
    }
});
```

L'EventDispatchThread, re

- seule thread à avoir le droit de modifier l'interface
- toute modification sur un composant graphique doit s'effectuer dans cette thread:
  - pas de problème quand on est dans le code d'un listener: on est dans la bonne thread
  - que faire quand ce n'est pas le cas ?

L'EventDispatchThread, re

- exemple: **`main`** veut mettre l'heure dans la barre de titre de l'application

- comment effectuer **`frame.setTitle(...)`** depuis une autre thread ?
- réponse: **`EventQueue.invoke...`**

EventQueue.invoke...

- `EventQueue.invokeLater(Runnable r)`:
  - crée un événement chargé d'exécuter `r.run()`
  - le poste dans la thread Swing
  - rend la main à la thread courante

- un peu lourd à écrire, mais c'est un mal nécessaire...

EventQueue.invoke...

- voici notre horloge:

```
...

/* We go on executing main's code in the main thread that
 * is NOT the event dispatch thread */
Calendar c=Calendar.getInstance();
while (true) {
    c.setTimeInMillis(System.currentTimeMillis());
    final String time=c.get(Calendar.HOUR_OF_DAY)+":"+c.get(Calendar.MINUTE)+":"
                          +c.get(Calendar.SECOND);
    try {
        EventQueue.invokeLater(new Runnable() {
            @Override
            public void run() {
                f.setTitle(time);
            }
        });
        Thread.sleep(1000);
    } catch (InterruptedException e) {
        throw new AssertionError(e); // should not occur
    }
}
```

EventQueue.invoke...

- `EventQueue.invokeAndWait(Runnable r)`:
  - même mode d'emploi, mais ne rend la main que quand l'événement a été traité
- utile:
  - quand on veut être sûr qu'une modification de l'interface a été réalisée
  - pour éviter de faire exploser la queue d'événements avec trop de posts

EventQueue.invoke...

- exemple qui explose la machine:

```
int i = 0;
while (true) {
    final int v = i++;
    EventQueue.invokeLater(new Runnable() {
        @Override
        public void run() {
            f.setTitle(v + "");
            l.setText("The event queue is overloaded with "+v+" invokeLater calls");
        }
    });
}
```

- pourquoi ? parce qu'il y a beaucoup trop d'événements à gérer

EventQueue.invoke...

- exemple qui n'explose pas la machine:

```
int i = 0;
while (true) {
    final int v = i++;
    try {
        EventQueue.invokeAndWait(new Runnable() {
            @Override
            public void run() {
                f.setTitle(v + "");
                l.setText("The event queue is not overloaded with "+v+" invokeAndWait");
            }
        });
    } catch (InterruptedException e) {
        throw new AssertionError(e); // should not occur
    } catch (InvocationTargetException e) {
        /* When a Runnable raises an exception in the EDT, it is wrapped into an
         * InvocationTargetException, so that we must unwrap it in the current thread */
        Throwable cause=e.getCause();
        if (cause instanceof RuntimeException) throw (RuntimeException)cause;
        if (cause instanceof Error) throw (Error)cause;
        throw new UndeclaredThrowableException(cause);
    }
}
```

- pourquoi ? un seul événement posté à la fois

Le Timer

- autre façon d'exécuter du code dans la thread Swing:
  - **`Timer(int delay,ActionListener toDo)`**
  - bien prendre le **`Timer`** de **`javax.swing`** !
- version timer de notre horloge:

```
final Calendar c=Calendar.getInstance();
new Timer(1000,new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        c.setTimeInMillis(System.currentTimeMillis());
        String time=c.get(Calendar.HOUR_OF_DAY)+":"+c.get(Calendar.MINUTE)+":"
                        +c.get(Calendar.SECOND);
        /* No need to use invoke... because we are in the event dispatch thread */
        f.setTitle(time);
    }
}).start();
```

Le rafraîchissement

- `repaint()` est threadsafe: elle poste un événement:
  - on peut donc l'utiliser même en dehors de la thread Swing
  - même chose pour `validate()` et `revalidate()`
- les `PaintEvent` générés par `repaint()` sont coalescents:
  - s'il y en a déjà un dans la queue d'événements, l'EDT n'en reposte pas un nouveau, afin de ne pas ralentir l'application

Le rafraîchissement

- exemple: un label personnalisé qui met du temps à se dessiner et qui compte les rafraîchissements

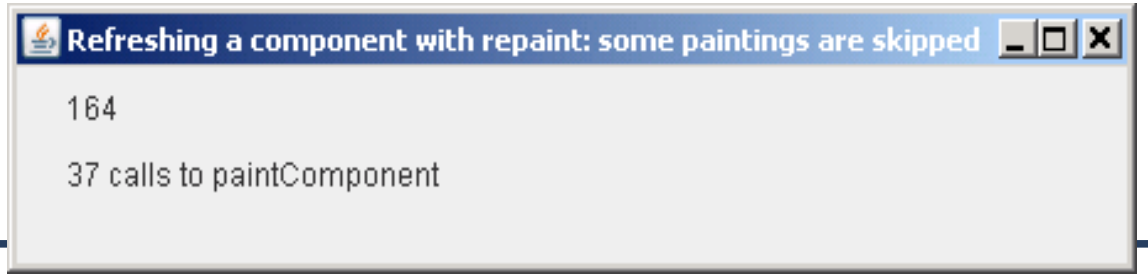
```
@Override
protected void paintComponent(Graphics g) {
    if (text!=null) {
        g.drawString(text,20,20);
    }
    counter++;
    g.drawString(counter+" call"+(counter>1?"s":"")+" to paintComponent",20,45);
    /* We wait to simulate a long painting */
    try {
        Thread.sleep(500);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

Le rafraîchissement

- si l'on poste beaucoup de **`repaint()`**:

```
int i=0;
while (true) {
    final int n=i++;
    EventQueue.invokeLater(new Runnable() {
        @Override
        public void run() {
            l.setCaption(n+"");
            l.repaint();
        }
    });
    try {
        Thread.sleep(100);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

- certains **`PaintEvent`** sont ignorés:



Le rafraîchissement

- mais, si l'on remplace **`repaint()`** par **`paintImmediately`** qui s'exécute sans attendre:

```
EventQueue.invokeLater(new Runnable() {
    @Override
    public void run() {
        l.setCaption(n+"");
        l.paintImmediately(l.getBounds());
    }
});
```

- tous les rafraîchissements ont lieu et l'application est plus lente:

Le SwingWorker

- problème: que faire quand une thread doit faire beaucoup d'opérations répétitives dans l'interface ?
  - `invokeAndWait` va ralentir la thread de calcul
  - `invokeLater` risque de surcharger la queue d'événements
- solution: le `SwingWorker` qui permet de regrouper automatiquement des paquets d'opérations

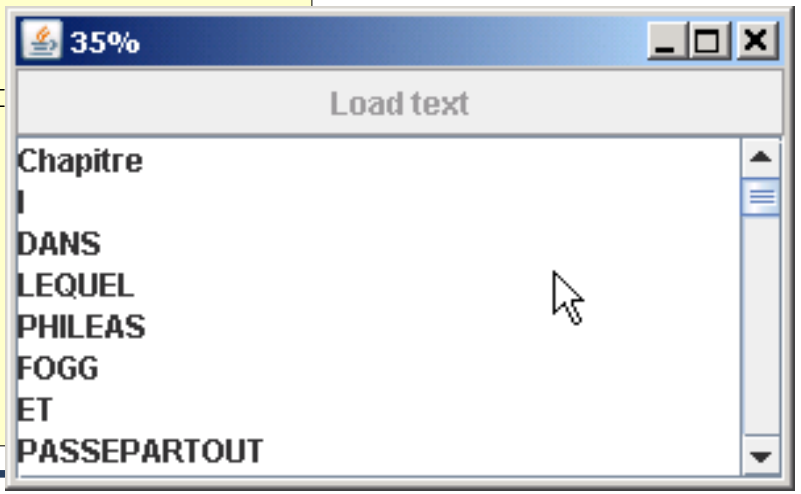
Le SwingWorker

- principe:
  - on travaille dans la méthode **`doInBackground`**
  - on poste des objets représentant des "résultats", grâce à la méthode **`publish`**
  - régulièrement, la thread Swing exécute la méthode **`process`** en lui passant tous les objets résultats qui n'ont pas encore été traités; libre alors au programmeur de faire ses rafraîchissements en fonction des résultats

Le SwingWorker

- exemple: chargement des mots d'un texte dans une liste

```
/* We have nothing special to return, so we use
 * the special type Void, which is an object version
 * of void; String is the type of our intermediate results */
SwingWorker<Void,String> worker=new SwingWorker<Void,String>() {
    @Override protected Void doInBackground() throws Exception {
        URL url=SwingWorkerDemo.class.getResource("80jours.txt");
        long sizeInBytes=FileUtilities.getSize(url);
        CounterInputStream stream=new CounterInputStream(url.openStream());
        Scanner scanner=new Scanner(stream,"UTF-16LE");
        /* We look for letter sequences */
        scanner.useDelimiter("[^\\p{javaLowerCase}\\p{javaUpperCase}]");
        while (scanner.hasNext()) {
            publish(scanner.next());
            int old=getProgress();
            setProgress((int)(stream.getCounter()*100./sizeI
            if (old!=getProgress()) {
                /* If we don't slow down, the
                 * progression won't be visible */
                Thread.sleep(1);
            }
        }
        stream.close();
        return null;
    }
```



Le SwingWorker

```
@Override
protected void process(List<String> chunks) {
    /* The only thing we have to do here is to add each
     * String to our list model */
    for (String s:chunks) {
        model.addElement(s);
    }
}

@Override
protected void done() {
    b.setEnabled(true);
    f.setTitle("SwingWorker demonstration");
}
```

- **attention:** `done` est exécutée quand `doInBackground` est finie, pas quand tous les résultats ont été traités par `process` !

Le SwingWorker

- on démarre le `SwingWorker` avec `execute`
- on ne se ressert pas d'un `SwingWorker`, on le recrée
- si on veut récupérer la valeur de retour de `doInBackground`, utiliser la méthode `get`
- attention: `get` est bloquant
  - ne pas l'appeler depuis la thread Swing!

Le SwingWorker

- on peut tenir à jour la progression avec **`setProgress(int n)`** avec **`n`** entre 0 et 100
- exemple: gestion de la progression

```
SwingWorker<Void,String> worker=new SwingWorker<Void,String>() {
...
};
worker.addPropertyChangeListener(new PropertyChangeListener() {
    @Override
    public void propertyChange(PropertyChangeEvent evt) {
        if ("progress".equals(evt.getPropertyName())) {
            f.setTitle((Integer)evt.getNewValue()+"%");
        }
    }
});
worker.execute();
```